Design Pattern - The Challenge

Below is a brief outline of how each design pattern challenge can be integrated into the app.

1. Singleton Pattern Challenge:

Scenario:

Your **MusicFinder** application needs a logger to track search queries for artist names and song titles. Implement this logger using the **Singleton Pattern** so that only one logger instance is created and shared across all the controllers.

Instructions:

- Create a `Logger` class that will log every search query.
- Ensure that this `Logger` follows the **Singleton Pattern**, meaning it must have:
  - A private constructor.
  - A static method to return the single instance.
  - Synchronisation for thread safety.

Hint:

- You need to use this logger instance inside the `MusicFinderController` to log all search requests (artist and song titles).

▼ **Solution:**

We will add a `Logger` class to the `MusicFinder` app to log search queries (artist and song titles). The logger should be implemented as a Singleton to ensure only one instance exists across the application.

Step 1: `Logger` Class

```
public class Logger {

    // Private static instance to hold the single instan
ce of Logger
    private static Logger instance;

    // Private constructor to prevent instantiation
    private Logger() {}

    // Public method to provide global access to the ins
tance
    public static synchronized Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    // Log message method
    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

Step 2: Integrate Logger

Inside the `MusicFinderController`, we will integrate the `Logger` to log every search query made by the user.

```
package com.example.musicfinder;

import org.springframework.web.bind.annotation.GetMappin
```

```
g;
import org.springframework.web.bind.annotation.RequestPa
ram;
import org.springframework.web.bind.annotation.RestContr
oller;

@RestController
public class MusicFinderController {

    @GetMapping("/search")
    public String findMusic(@RequestParam String artist,
@RequestParam String song) {
        // Log the search query
        Logger.getInstance().log("Searching for: " + art
ist + " - " + song);

        // Existing logic to search for music
        // For example, get lyrics or YouTube results
        return "Results for: " + artist + " - " + song;
    }
}
```

---

2. Factory Method Challenge:

Scenario:

Your Music Finder App is expanding to include different search providers (e.g., YouTube, Spotify). Implement a **Factory Method** that generates search objects for different music platforms.

Instructions:

- Create an abstract `MusicSearch` class that defines a `search` method.
- Implement concrete classes like `YouTubeSearch`, `SpotifySearch` etc.
- Implement a `SearchFactory` class that will return the correct search object based on the platform.

Hint:

You need to extend the application so it can dynamically decide which platform to use for the search.

▼ **Solution:**

We will create a factory that can instantiate different search providers like **YouTube** and **Spotify**. This will allow the user to choose a provider for their search query.

Step 1: Define an Abstract MusicSearch Class

```
abstract class MusicSearch {
    public abstract String search(String artist, String
song);
}
```

Step 2: Implement Specific Search Classes (YouTube, Spotify, etc.)

```
class YouTubeSearch extends MusicSearch {
    @Override
    public String search(String artist, String song) {
        // Implement YouTube search logic
        return "YouTube search result for: " + artist +
" - " + song;
    }
}

class SpotifySearch extends MusicSearch {
    @Override
    public String search(String artist, String song) {
        // Implement Spotify search logic
        return "Spotify search result for: " + artist +
" - " + song;
    }
}
```

Step 3: Create a `SearchFactory`

```
class SearchFactory {
    public MusicSearch getSearchProvider(String provide
```

```
r) {
        if (provider.equalsIgnoreCase("YouTube")) {
            return new YouTubeSearch();
        } else if (provider.equalsIgnoreCase("Spotify"))
{
            return new SpotifySearch();
        }
        return null;
    }
}
```

Step 4: Integrate into `MusicFinderController`

```
package com.example.musicfinder;

@RestController
public class MusicFinderController {

    @GetMapping("/search")
    public String findMusic(@RequestParam String artist,
@RequestParam String song, @RequestParam String provide
r) {
        // Log the search query
        Logger.getInstance().log("Searching for: " + art
ist + " - " + song + " on " + provider);

        // Use the factory to get the right provider
        SearchFactory factory = new SearchFactory();
        MusicSearch search = factory.getSearchProvider(p
rovider);

        if (search != null) {
            return search.search(artist, song);
        } else {
            return "Invalid provider";
        }
    }
}
```

3. Adapter Pattern Challenge:

Scenario:

The existing **MusicFinderController** uses an external lyrics provider, but now we want to integrate a legacy lyrics API that provides lyrics in a different format. You need to implement an **Adapter Pattern** to adapt the legacy API output to the current format.

Instructions:

- Implement an adapter to convert the legacy API's lyrics format into the format expected by your app.

Hint:

- The adapter should take the output from the legacy lyrics provider and transform it into the JSON format currently expected by the MusicFinder application.

▼ **Solution:**

We will implement an adapter for a legacy lyrics API that returns data in an incompatible format.

Step 1: Legacy Lyrics Service

```
class LegacyLyricsService {
    public String getLyricsLegacy(String artist, String
song) {
        return "Legacy lyrics for: " + artist + " - " +
song;
    }
}
```

Step 2: Implement the Adapter

```
class LyricsAdapter {
    private LegacyLyricsService legacyService;

    public LyricsAdapter(LegacyLyricsService legacyServi
ce) {
        this.legacyService = legacyService;
```

```
    }

    public String getFormattedLyrics(String artist, Stri
ng song) {
        String legacyLyrics = legacyService.getLyricsLeg
acy(artist, song);
        return formatToJSON(legacyLyrics);
    }

    private String formatToJSON(String legacyLyrics) {
        // Format the legacy lyrics to JSON format
        return "{ \"lyrics\": \"" + legacyLyrics + "\"
}";
    }
}
```

Step 3: Integrate into `MusicFinderController`

```
package com.example.musicfinder;

@RestController
public class MusicFinderController {

    @GetMapping("/lyrics")
    public String getLyrics(@RequestParam String artist,
@RequestParam String song) {
        // Use the adapter to get lyrics from the legacy
service
        LyricsAdapter adapter = new LyricsAdapter(new Le
gacyLyricsService());
        return adapter.getFormattedLyrics(artist, song);
    }
}
```

---

4. Decorator Pattern Challenge:

Scenario:

Users can add different effects to their search results, such as filtering by language or adding a popularity score. Implement the **Decorator Pattern** to dynamically apply these effects.

Instructions:

- Create a base `SearchResult` class.
- Implement decorators like `LanguageFilterDecorator` and `PopularityScoreDecorator` that extend the functionality of the search result.

Hint:

- The decorators should dynamically add functionality to the base `SearchResult` class.

▼ **Solution**:

We can now apply the **Decorator Pattern** to dynamically add options (e.g., filters or tags) to search results. In this case, let's dynamically modify search results.

Step 1: Basic `SearchResult` Component

```
class SearchResult {
    public String getResult() {
        return "Basic search result";
    }
}
```

Step 2: Decorator Implementation

```
abstract class SearchResultDecorator extends SearchResul
t {
    protected SearchResult decoratedResult;

    public SearchResultDecorator(SearchResult decoratedR
esult) {
        this.decoratedResult = decoratedResult;
    }

    public String getResult() {
        return decoratedResult.getResult();
```

```
    }
}

class FilteredByLanguageDecorator extends SearchResultDe
corator {
    public FilteredByLanguageDecorator(SearchResult deco
ratedResult) {
        super(decoratedResult);
    }

    public String getResult() {
        return super.getResult() + " [Filtered by langua
ge]";
    }
}

class PopularityScoreDecorator extends SearchResultDecor
ator {
    public PopularityScoreDecorator(SearchResult decorat
edResult) {
        super(decoratedResult);
    }

    public String getResult() {
        return super.getResult() + " [With popularity sc
ore]";
    }
}
```

Step 3: Integrate into `MusicFinderController`

```
package com.example.musicfinder;

@RestController
public class MusicFinderController {

    @GetMapping("/search")
    public String findMusic(@RequestParam String artist,
```

```
@RequestParam String song) {
        SearchResult searchResult = new SearchResult();

        // Add decorators
        searchResult = new FilteredByLanguageDecorator(s
earchResult);
        searchResult = new PopularityScoreDecorator(sear
chResult);

        return searchResult.getResult();
    }
}
```